Stretching Multi-Ring Paxos

Samuel Benz Leandro Pacheco de Sousa Fernando Pedone
University of Lugano (USI), Switzerland

***Abstract*—Internet-scale services rely on data partitioning and replication to provide scalable performance and high availability. Moreover, to reduce user-perceived response times and tolerate disasters (i.e., the failure of a whole datacenter), services are increasingly becoming geographically distributed. Data partitioning and replication, combined with local and geographical distribution, introduce daunting challenges, including the need to carefully order requests among replicas and partitions. One way to tackle this problem is to use group communication primitives that encapsulate order requirements. This paper presents a detailed performance evaluation of Multi-Ring Paxos, a scalable group communication primitive. We focus our analysis on "extreme conditions" with deployments including high-end 10 Gbps networks, a large number of combined rings (i.e., independent Paxos instances), a large number of replicas in a ring, and a global deployment. We also report on the performance of recovery under peak load and present two novel extensions to boost Multi-Ring Paxos's performance.**

I. Introduction

Internet-scale services are widely deployed today. These systems must deal with a virtually unlimited user base, scale with high and often fast demand of resources, and be always available. In addition to these challenges, many current services have become geographically distributed. Geographical distribution helps reduce user-perceived response times and increase availability in the presence of node failures and datacenter disasters (i.e., the failure of an entire datacenter). In these systems, data *partitioning* (also known as sharding) and *replication* play key roles.

Data partitioning and replication can lead to highly scalable and available systems, however, they introduce daunting challenges. Handling partitioned and replicated data has created a dichotomy in the design space of large-scale distributed systems. One approach, known as *weak consistency*, makes the effects of data partitioning and replication visible to the application. Weak consistency provides more relaxed guarantees and make systems less exposed to impossibility results [1], [2]. The tradeoff is that weak consistency generally leads to more complex and less intuitive applications. The other approach, known as *strong consistency*, hides data partitioning and replication from the application, simplifying application development. Strong consistency requires ordering requests across the system in order to provide applications with the illusion that state is neither partitioned nor replicated.

Reliably delivering and ordering requests in a distributed system has been extensively studied in the context of group communication (e.g., [3], [4]). Atomic broadcast and atomic multicast, for example, encapsulate the notions of totally and partially ordering requests in a distributed system, respectively. Among the many group communication protocols proposed in the literature [5], this paper focuses on Multi-Ring Paxos [6], [7], an atomic multicast protocol based on Paxos [8]. Multi-Ring Paxos was designed to scale throughput with the addition of resources, a characteristic uncommon to group communication systems. Existing atomic broadcast and multicast implementations are typically bounded by the capacity of the nodes that take part in the ordering protocol. Increasing the number of nodes improves availability, but not performance. Multi-Ring Paxos scales throughput by composing multiple independent instances of Paxos (i.e., *rings*). Distributing the load among independent Paxos instances is important to cope with CPU bottlenecks (e.g., as typically happens with the coordinator in Paxos [9]) and I/O bottlenecks (e.g., acceptor's disks [6]).

Multi-Ring Paxos has been shown to perform well in locally and geographically distributed settings [6], [7]. In this study, we set out to assess its performance under extreme conditions. In addition to deepening our understanding about Multi-Ring Paxos, the study also resulted in a number of performance optimizations, which we describe in the following sections. Our performance assessment was guided by our desire to answer the following questions.

- Can Multi-Ring Paxos deliver performance that matches high-end networks (i.e., 10 Gbps)?
- How does a recovering replica impact the performance of operational replicas computing at peak load?
- Multi-Ring Paxos ensures high performance despite unbalanced load in combined rings with a skip mechanism. Can Multi-Ring Paxos's skip mechanism handle highly skewed traffic?
- How many combined rings in a learner and learners in a ring are "too many"?
- Can Multi-Ring Paxos deliver usable performance when deployed around the globe and subject to disasters?

This paper makes the following contributions. First, we review Multi-Ring Paxos's design and introduce two novel techniques, *latency compensation* and *non-disruptive recovery*, which improve Multi-Ring Paxos's performance under strenuous conditions. Second, we detail Multi-Ring Paxos implementation, evaluate its performance experimentally and answer all questions raised above. Third, we discuss the lessons we learned during the implementation and evaluation of Multi-Ring Paxos.

The rest of the paper is structured as follows. Section II presents the design of Multi-Ring Paxos and two techniques that improve its performance. Section III evaluates the performance of Multi-Ring Paxos. Section IV reviews related work. Section V discusses our experiences with Multi-Ring Paxos and Section VI concludes the paper.

II. Multi-Ring Paxos

In this section we introduce assumptions and definitions used in Multi-Ring Paxos (Section II-A), describe the protocol in detail (Section II-B), present two novel optimizations to Multi-Ring Paxos (Sections II-C and II-D), and discuss the protocol's implementation (Section II-E).

A. System model and definitions

We assume a distributed system composed of interconnected processes that communicate through message passing. Processes may fail by crashing and subsequently recover, but do not experience arbitrary behavior (i.e., no Byzantine failures). Processes are either *correct* or *faulty*. A correct process is eventually operational "forever" and can reliably exchange messages with other correct processes. In practice, "forever" means long enough for processes to make some progress (e.g., terminate one instance of consensus).

Multi-Ring Paxos, like Paxos [10], ensures safety under both asynchronous and synchronous execution periods. To ensure liveness, we assume the system is *partially synchronous* [11]: it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous, called the *Global Stabilization Time (GST)* [11], is unknown to the processes. Before GST, there are no bounds on the time it takes for messages to be transmitted and actions to be executed. After GST, such bounds exist but are unknown.

Atomic multicast is a communication abstraction defined by the primitives *multicast*(γ, m) and *deliver*(m), where m is a message and γ is a multicast group. Processes choose from which multicast groups they wish to deliver messages. If process p chooses to deliver messages multicast to group γ, we say that p *subscribes to* group γ. Let relation $<$ be defined such that $m < m'$ iff there is a process that delivers m before m'. Atomic multicast ensures that (i) if a process delivers m, then all correct processes that subscribe to γ deliver m *(agreement)*; (ii) if a correct process p multicasts m to γ then all correct processes that subscribe to γ deliver m *(validity)*; and (iii) relation $<$ is acyclic *(order)*. Atomic broadcast is a special case of atomic multicast where there is a single group to which all processes subscribe.

B. Protocol design

Multi-Ring Paxos uses independent Ring Paxos [9] instances to implement atomic multicast, where each Ring Paxos instance corresponds to an atomic multicast group, as defined in the previous section (see Figure 1). Ring Paxos efficiently implements the Paxos algorithm [10] by disposing processes in a ring overlay—for this reason, we sometimes refer to a Ring Paxos instance as a *ring*. Multi-Ring Paxos ensures ordered delivery of messages using a deterministic round-robin mechanism to merge messages ordered by different rings into a single stream of messages [6].

Multi-Ring Paxos's deterministic merge mechanism ensures that any two messages delivered by two or more processes are delivered in the same order. If no additional precautions are observed, however, a process will deliver messages at the pace of the slowest multicast group it subscribes to. To handle unbalanced traffic among rings, a slow ring can skip a configurable number of messages to keep up with the pace of faster rings. The number of "skip messages" in a ring is determined based on a virtual maximum throughput λ that the fastest ring can achieve, measured in messages per second. Periodically, each ring coordinator, co-located with its Paxos leader, calculates the number of required skip messages to reach λ since the last calculation and multicasts these messages in its ring. Upon delivering a skip message, a process simply discards it. Multiple skip messages are grouped in a single Paxos round, containing the number of messages to be skipped.

The number of messages to be skipped in a ring at time t_{now}, denoted $skips(t_{now})$, is calculated using a referential time t_{ref} and the total number of messages already skipped in the ring, denoted $skipped$, as shown next.

$$skips(t_{now}) = \lambda * (t_{now} - t_{ref}) - skipped \qquad (1)$$

Time t_{ref} can be set to 0 or the system's start up time, but it must be the same for the coordinators of all rings. In order to even out the throughput of the various rings, so that the merge mechanism is effective, every coordinator should periodically recompute equation (1). In our experiments, we synchronize the coordinator's clocks using a simple NTP service, so that coordinators choose approximately the same t_{now} when recomputing equation (1). Note that clock accuracy does not affect the correctness of Multi-Ring Paxos, but has an impact on its performance.

C. Latency compensation

The skip calculation described in the previous section is very effective in networks subject to small latencies (e.g., within a datacenter). However, with large and disparate latencies (e.g., geographical deployments), a late skip message may delay the delivery of messages at a learner (see Figure 1(c)). This delay might happen even if the number of skip instances is accurately calculated to account for unbalanced traffic among rings. We overcome this problem by revisiting the skip mechanism to take into consideration the approximate time skip messages need to reach their concerned learners. In equation (2), avg_delay is an approximated average of the delays between the ring coordinator and the ring learners. The intuition is to skip

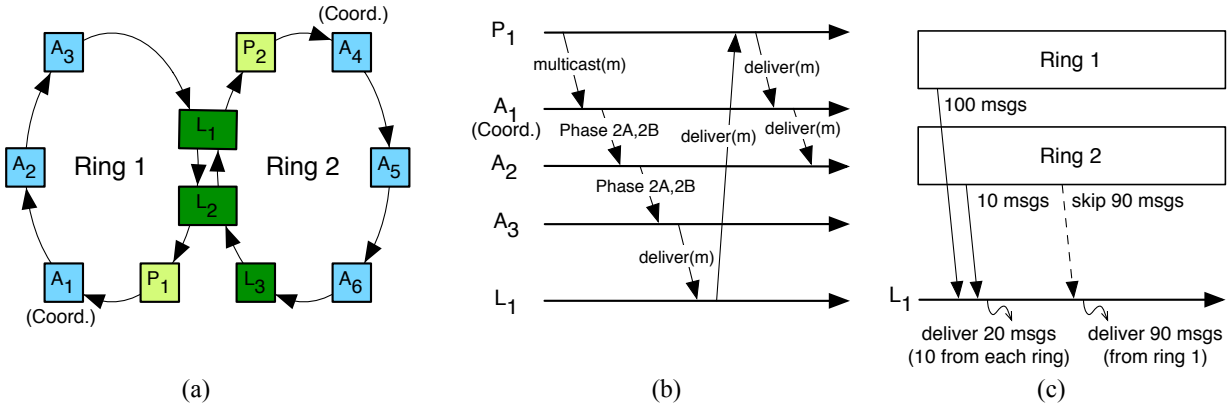

Figure 1. (a) The various process roles in Ring Paxos disposed in two rings (learners L_1 and L_2 deliver messages from Rings 1 and 2, and leaner L_3 delivers messages from Ring 2 only); (b) an execution of a single instance of Ring Paxos; and (c) Multi-Ring Paxos's skip mechanism.

additional messages to make up for the time it takes for a skip message to arrive at the learners.

$$skips(t_{now}) = \lambda * (t_{now} - t_{ref} - avg\_delay) - skipped \quad (2)$$

D. Non-disruptive recovery

Recovering a failed learner in Multi-Ring Paxos, as described in [7], boils down to (a) retrieving and installing the most recent service's checkpoint and (b) recovering and executing commands that are not included in the retrieved snapshot, the *log tail*. While this procedure can be optimized in many ways [12], recovery in Multi-Ring Paxos is inherently subject to a tradeoff that involves the frequency of checkpoints and the size of the log tail: frequent checkpoints result in smaller log tails and, conversely, infrequent checkpoints lead to larger log tails.

Since checkpoints tend to slow down service execution, reducing the frequency of checkpoints seems desirable. However, restricting the log tail size is equally important because retrieving commands from the log during recovery has negative effects on the service's performance—this happens because acceptors must participate in new rounds of Paxos and at the same time retrieve values accepted in earlier rounds (i.e., the log tail). We have experimentally assessed that even under moderate load the recovery traffic drastically affects performance (see Section III-E).

To minimize disruption of service performance during normal service execution and recovery of a learner, we revisited Multi-Ring Paxos's original recovery mechanism [7]. With the new method, a recovering learner starts by caching new ordered messages. This silent procedure does not place acceptors under additional stress. The replica then must retrieve a valid checkpointed state from another replica (or from remote storage), that is, a checkpoint that contains all commands that precede the cached commands. With a valid checkpoint, the replica can apply the cached commands not in the the checkpoint and discard the ones already in the checkpoint. This procedure prioritizes performance during normal operation but it may increase the time needed to recover a learner.

E. Implementation

URingPaxos,[1] our Multi-Ring Paxos prototype, is entirely implemented in Java with communication relying on TCP. The ring and configuration management are handled by Zookeeper [13]. A URingPaxos node can play multiple Paxos roles (e.g., proposer, acceptor, learner). Upon starting, the node registers its IP address and its intended roles with Zookeeper. The node is then informed by Zookeeper about the endpoints it should connect to to form a ring. If a node crashes, Zookeeper will inform all nodes about the topology change. Acceptors have access to stable storage. Depending on the required guaranties, back-ends for in-memory storage and synchronous and asynchronous on-disk storage are available. The first acceptor in the ring is elected the coordinator.

We implemented our own internal serialization based on byte buffers and ensure that at most one object is created per received item. To avoid heavy garbage collection work, the current implementation pre-allocates an array of byte buffers. As a result, under normal load, garbage collection does not significantly impact performance.

III. Experimental Evaluation

In this section, we explain our goals and methodology, describe our experimental setup, detail Multi-Ring Paxos configuration, and present and comment our findings.

A. Objectives and methodology

We aim to assess the behavior of Multi-Ring Paxos under a range of "extreme" conditions, including wide-area channels and high-performance links. Since we do not have access to an experimental environment that simultaneously

[1] https://github.com/sambenz/URingPaxos

accommodates all these characteristics, we conducted our experiments in different environments and workload settings, as described next.

- We scale the number of rings to achieve high performance in a high-end 10 Gbps network (Section III-D).
- We evaluate the impact of a recovering replica on the performance of operational replicas under peak load (Section III-E).
- We stress Multi-Ring Paxos skip mechanism with highly skewed traffic (Section III-F).
- We study the impact of many learners in a single ring (Section III-G).
- We assess the impact of a global ring and a disaster failure in a geographically distributed deployment (Section III-H).

B. Experimental setup

The local-area network experiments (i.e., within a datacenter) were performed in two environments: (a) A cluster of 4 servers equipped with 32-core 2.6 GHz Xeon CPUs and 128 GB of main memory. These servers were interconnected through a 48-port 10-Gbps switch with round trip time of 0.1 millisecond. (b) A cluster of 24 Dell PowerEdge 1435 servers and 40 HP SE1102 servers connected through two HP ProCurve 2910 switches with 1-Gbps interfaces. The globally distributed experiments (i.e., across datacenters) were performed on Amazon EC2 with instances in 5 different regions. We used r3.large spot-instances, with 2 vCPU and 15 GB DRAM. To avoid disk bottlenecks, all experiments were executed with in-memory storage. A detailed evaluation of Multi-Ring Paxos under different storage conditions can be found in [7].

C. Multi-Ring Paxos configuration

Multi-Ring Paxos has three configuration parameters [6]: M, λ and Δ_t. M is the number of messages delivered (or skipped) contiguously from the same single ring; if not stated otherwise, we use $M = 1$.

We have empirically determined that λ, the virtually maximum throughput of a ring, should be set a bit higher than the actual maximum achievable performance. Too high λ values lead to wasted CPU cycles in the deterministic merge function; too low λ values cap performance.

Parameter Δ_t determines how often skip messages are proposed in a Paxos instance. In general, small values for Δ_t are preferred, to reduce the latency of actual messages; too low Δ_t values, however, waste Paxos instances and introduce additional overhead in the system.

D. Scaling up in a local 10 Gbps network

In this section, we evaluate the scalability of Multi-Ring Paxos in a local 10 Gbps network environment.

Setup. We perform two sets of experiments, one with 200-byte messages and another with 32-Kbyte messages. For each message size, we increase the number of rings from 1 (i.e., Ring Paxos) up to 10. Four servers are involved: one server runs one proposer and one acceptor per ring, two other servers play the role of acceptors only, with one acceptor deployed per ring; the last server runs a learner, which subscribes to up to 10 rings. The proposer in each ring uses multiple threads (20), one thread per client. We report peak throughput, measured at the learner.

Results. Figures 2 and 3 (top left graphs) show, respectively, that Multi-Ring Paxos reaches peak performance with 9 rings for large messages and with 8 rings for small messages. With large messages, Multi-Ring Paxos reaches 8.41 Gbps, very close to 8.75 Gbps, the maximum usable TCP throughput (i.e., without TCP/IP headers) we could produce with *iperf*.[2] With small messages, Multi-Ring Paxos achieves about 570 K messages per second. We also report the latency CDF, measured in 1-millisecond buckets, for the peak load (top center graphs) and the CPU consumption at the learner (top right graphs). The 90-th latency percentile under these conditions is below 5 milliseconds. The protocol is network-bound with large messages and CPU-bound with small messages. (Since there is one communication thread per ring at the learner, 10 rings can use up to 1000% CPU.)

In both experiments we can see (top left graphs) that as the number of rings a learner subscribes to increases, the throughput achieved by each ring decreases. This happens because the load in the learner's Java virtual machine increases with each new ring, slowing down the learner. In Multi-Ring Paxos, a slow process reduces the overall traffic, as a result of flow control. This effect can be seen in the garbage collection logs (bottom two graphs) of the two runs.

E. Non-disruptive recovery under peak load

To evaluate our optimized recovery procedure, we deploy a simple key-value store service, implemented on top of Multi-Ring Paxos [7]. Our key-value store service implements commands to insert and remove tuples of arbitrary size, read and update an existing entry, and query a range of tuples. Replicas use a copy-on-write data structure to allow checkpoints in parallel with the execution of commands.

Setup. The experimental setup uses a ring with 3 nodes, each acting as an acceptor and a learner (i.e., replica). Four clients (each with 150 threads) submit 1024-byte update requests to the replicas through Multi-Ring Paxos [14]. Each replica executes every request and replies back to the client using UDP. Every replica periodically checkpoints its state into a distributed file system,[3] accessible to all replicas. The state checkpointed by a replica has 1.5 million entries.

Results. Figure 4 shows the behavior of Multi-Ring Paxos's new non-disruptive recovery under maximum load, which for 1024-byte values is around 800 Mbps. For comparison, we also depict the behavior of the old recovery

[2] http://iperf.sourceforge.net/

[3] http://www.xtreemfs.org/

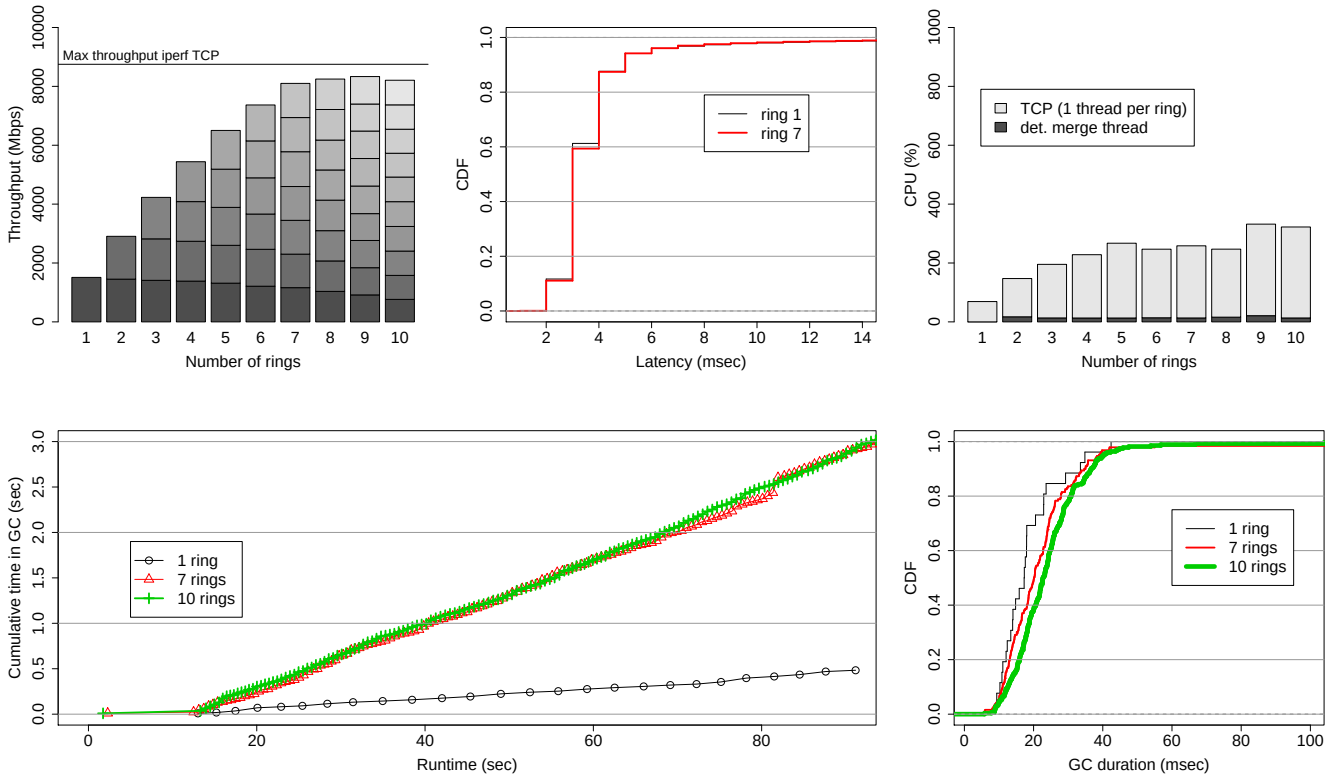

Figure 2. Scaling up Multi-Ring Paxos in a 10 Gbps network. The graphs show the aggregate and per ring throughput in megabits per second for 32-Kbyte messages (top left); the latency CDF, measured in 1-millisecond buckets (top center); the CPU usage (top right); garbage collection activity during some executions (bottom left) and the CDF of the duration of the Java garbage collection work (bottom right). All measurements performed at the learner process.

protocol under lower load, around 400 Mbps, since the old protocol cannot sustain higher load. Around 45 seconds into the execution, we crash one of the replicas, which starts recovery around time 110. With the new recovery protocol, the average throughput during recovery is 78% of the throughput under normal operation. Performance troughs are due to garbage collection (events labelled "1" in the graph) and ring management (event with label "2"). Since processes communicate in a ring, a pause in any of the nodes (e.g., due to garbage collection) can have a visible effect on throughput. The fact that the recovering learner has to batch new commands and that replicas have to use multiple (in-memory) copy-on-write data structures forces us to use large heaps, which lead to longer and unpredictable garbage collection pauses (see Section V).

F. The skip mechanism under highly skewed traffic

In Multi-Ring Paxos, learners can subscribe to any combination of existing rings. Unbalanced traffic across rings is compensated with the skip mechanism. In this experiment, we assess the overhead of the skip mechanism on highly skewed traffic.

Setup. This experiment was conducted in a local cluster with a 1 Gbps network. In this experiment, a single learner subscribes to multiple rings. Each ring is composed of three acceptors and the learner. In order to assess the protocol's inherent latency without any queuing effects, we consider executions with a single client. We varied the number of rings from 1 up to 32. Except for the configurations with 16 and 32 rings, we deploy one acceptor per node. For the experiments with 16 rings, there are two acceptors per node; with 32 rings, there are four acceptors per node. To assess the efficacy of the skip mechanism, the client submits 200-byte messages to one of the rings; the other rings rely solely on the skip mechanism. In these experiments, λ was set to 5 milliseconds.

Results. The most visible impact in Figure 5 is the transition from one to two rings. One ring is not constrained by any synchronization and can achieve the lowest latency. Additional rings introduce an overhead, that eventually increases linearly with the number of rings. Since we have one client only, from Little's law [15], the throughput is the inverse of the latency.

G. The performance of large rings

Although fault-tolerant deployments usually require a few replicas (e.g., three to five), having a large number of learners inside a single ring is also useful. One common example

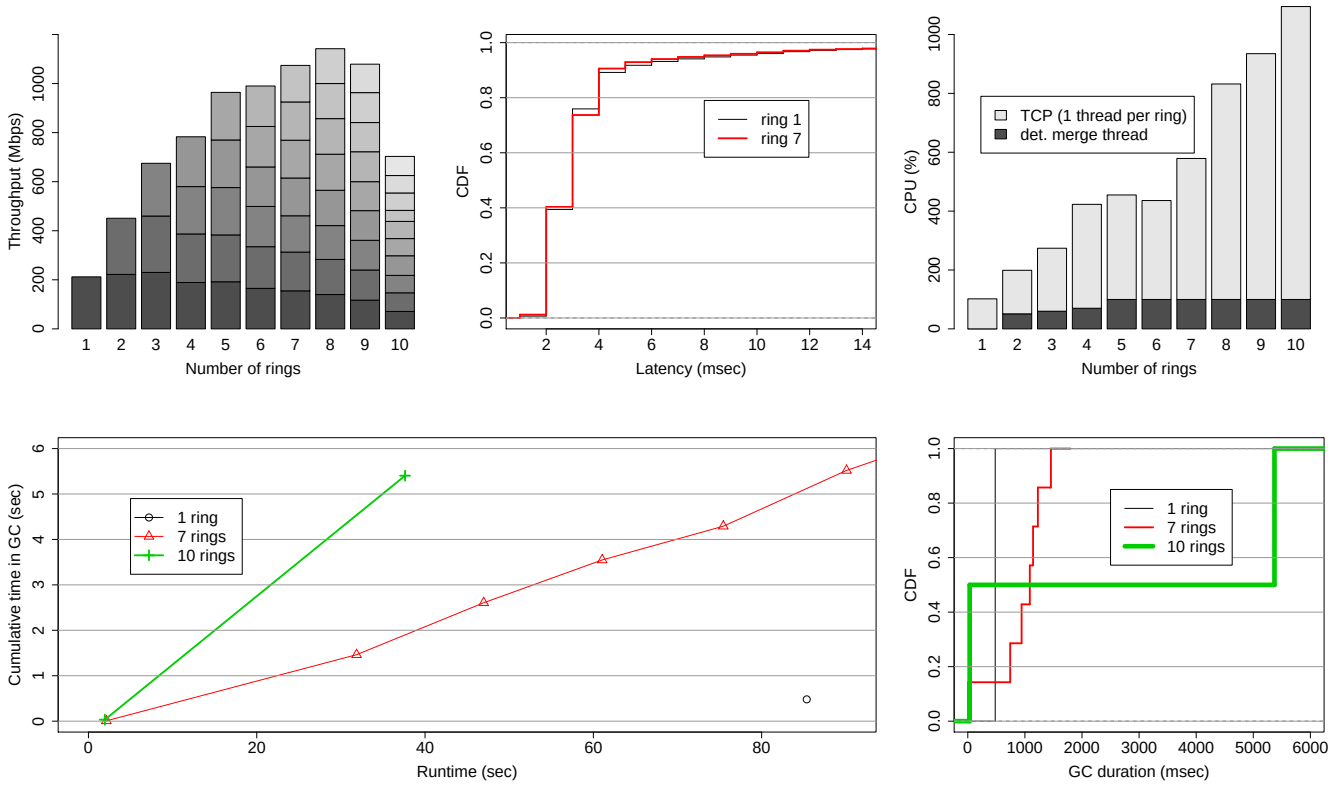

Figure 3. Scaling up Multi-Ring Paxos in a 10 Gbps network. The graphs show the aggregate and per ring throughput in megabits per second for 200-byte messages (top left); the latency CDF, measured in 1-millisecond buckets (top center); the CPU usage (top right); garbage collection activity during some executions (bottom left) and the CDF of the duration of the Java garbage collection work (bottom right). All measurements performed at the learner process.

is to use a large ring containing all replicas from smaller rings. In this case, each small ring would encompass some partition of the service's state and the large ring could be used to send commands concerning multiple partitions [7].

Setup. This experiment was conducted in a local cluster with a 1 Gbps network. There is one ring with three acceptors and an increasing number of learners. With up to 32 learners, we deployed each learner in an HP SE1102 server; we deployed additional learners in the weaker Dell PowerEdge servers. To assess the protocol's latency in the absence of any queuing effects due to contention, we consider executions with a single client, which sends messages to a proposer in the ring.

Results. Figure 6 shows the effect of adding learners to a single ring. Like in the previous experiment, the single client results in throughput inversely proportional to latency. Further, ring communication in Multi-Ring Paxos linearly adds latency with every additional node. The sharp bend in latency after adding 32 learners is caused by the weaker nodes, which become CPU-bound with small messages.

H. Ordering messages across the globe

In this section we evaluate the global scalability and fault tolerance of Multi-Ring Paxos. The goal is to show that having a large global ring, which allows to send ordered commands to geographically distributed partitions (local rings), does not slow down local traffic. We also evaluate the effect of a data center outage during runtime.

Setup. For this experiment, we used Amazon EC2 instances. We deployed 5 local rings, each in its own region: us-west-1 (N. California), us-west-2 (Oregon), eu-west-1 (Ireland), ap-southeast-1 (Singapore), ap-southeast-2 (Sydney). All nodes in each local ring are placed on the same availability zone. We also deployed a global ring, composed of three acceptors (placed in separate regions) and all learners from each of the local rings. This deployment allows for progress even in the presence of a disaster taking down an entire datacenter. We simulated a datacenter outage by forcibly killing all processes belonging to one of the regions containing an acceptor of the global ring.

Results. We first evaluate the fault tolerance of Multi-Ring Paxos. Figure 7 shows the throughput in each of the local rings, using messages of 32 Kbytes. We can see that, despite the outage of a complete region (at around 25 seconds of execution), the remaining rings maintain normal traffic after a short disruption caused by the global ring reconfiguration.

To assess the impact of a global ring on the performance

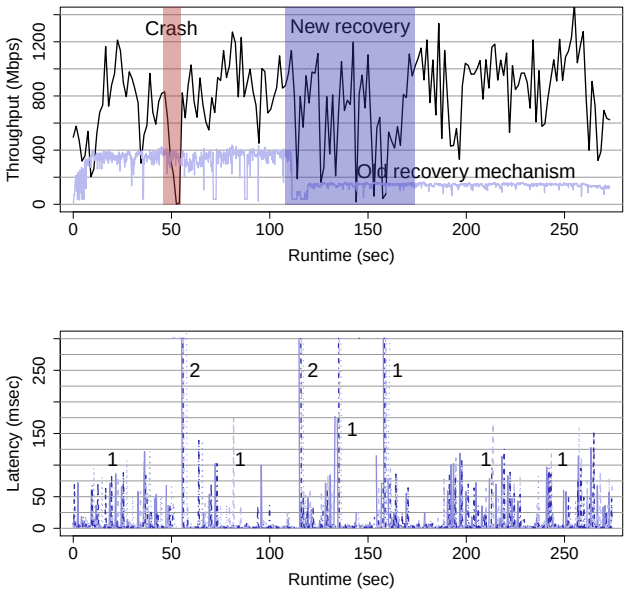

Figure 4. Recovery of a key-value store snapshot with 1.5 million entries. Throughput of Multi-Ring Paxos's new and old recovery protocols (top) and latency of new recovery protocol (bottom, where "1" identifies garbage collection events and "2" identifies ring management events).

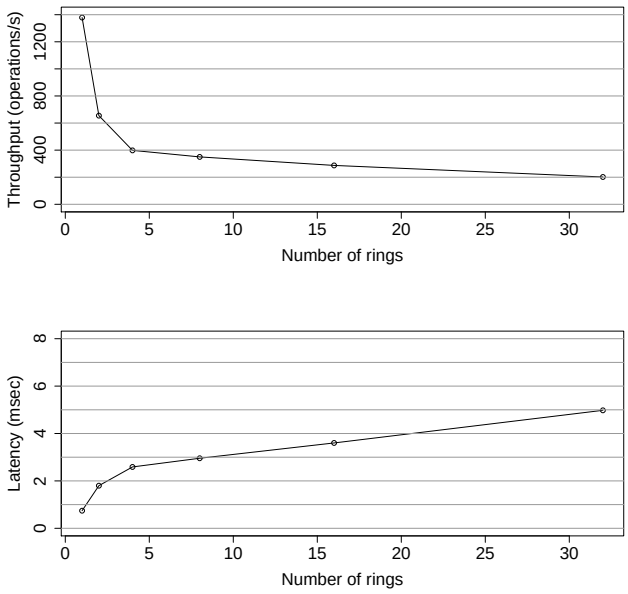

Figure 5. Impact of the number of groups (rings) a learner subscribes to on throughput and latency (since there is a single client, from Little's law throughput is the inverse of latency).

of local rings, we conducted a few other experiments using the same deployment of 5 datacenters, each with a local ring. We consider a baseline case with local rings only (i.e., no global ring) and setups with a global ring synchronizing all nodes, with and without latency compensation (Section II-C). We use the same load (number of clients) in all three cases, roughly 80% of the peak throughput for the case with compensation enabled, with 200-byte messages. Figure 8 shows the throughput obtained in each case and the latency CDF. The local throughput went down by around 23% with a global ring connecting all the nodes. The results also show that compensating the latency difference between rings is fundamental. The "steps" visible in the latency CDF for the scenario with no compensation reflect the latency difference across rings. Since the datacenters also form a (global) ring in the way they are arranged, the ordering and latency between them affects when a message from the global ring is delivered at each datacenter. Having the compensation mechanism allows for this latency difference to be masked.

IV. Related work

In this section, we briefly review related work on atomic multicast, geo-distributed systems, and high-performance recovery.

Atomic multicast: Atomic multicast has been extensively studied in the literature. In [3], a protocol is proposed for failure-free scenarios. To decide on the final timestamp of a message, each process in the set of message addressees locally chooses a timestamp, exchanges its chosen timestamps, deterministically agrees on one of them, and delivers messages according to the message's final timestamp. Several works have extended this algorithm to tolerate failures [16], [17], [18], [19], where the main idea is to replace failure-prone processes by fault-tolerant disjoint groups of processes, each group implementing the algorithm by means of state-machine replication.

Spread [20] implements a highly configurable group communication system, which supports the abstraction of process groups. Spread orders messages by the means of interconnected daemons that handle the communication in the system. Processes connect to a daemon to multicast and deliver messages. While the group abstraction is similar to the Totem Multi-Ring protocol [21], Totem uses timestamps to achieve global total order. Multi-Ring Paxos's deterministic merge strategy is similar to the work proposed in [22], which totally orders message streams in a widely distributed publish-subscribe system.

Geo-replication: There are different approaches to handling the high latency inherent of globally distributed systems. Some systems choose to weaken consistency guarantees (e.g., Dynamo [23]), while others cope with wide-area round trip times. Mencius [24] and EPaxos [25] are latency optimized. Both protocols implement atomic broadcast and therefore do not scale. P-store [26] relies on atomic multicast. In order to scale, it partitions the service state

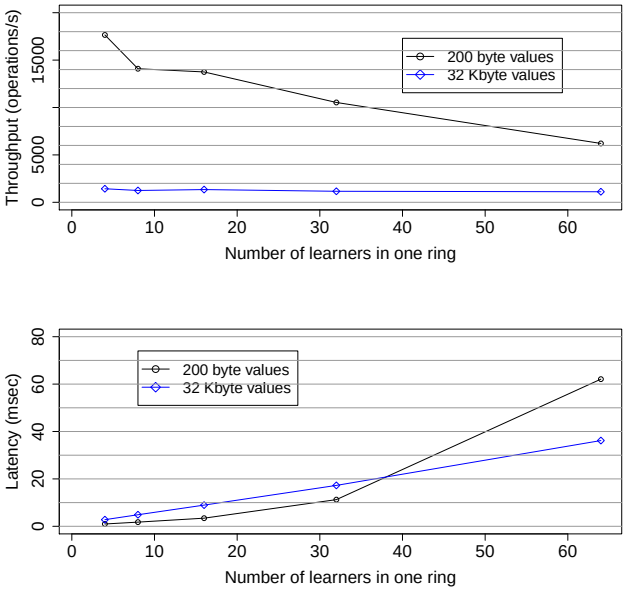

Figure 6. Impact of adding many learners to a single ring. Every node in the ring adds linearly latency.

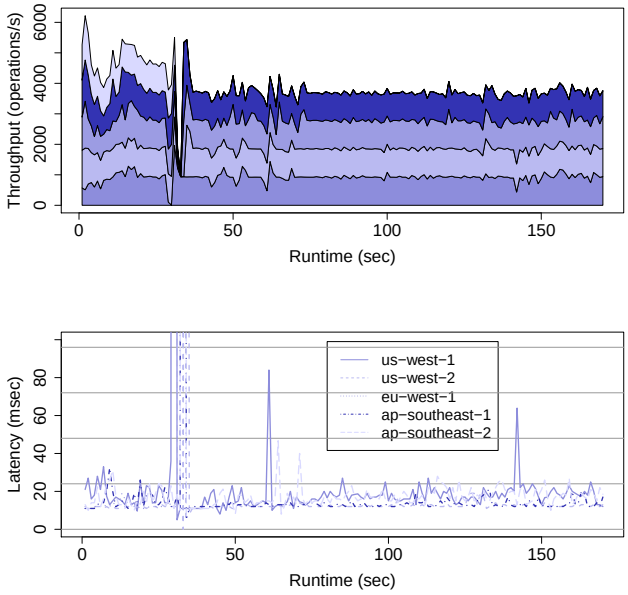

Figure 7. Impact of a data center outage after 25s of execution in the performance of a global Multi-Ring Paxos deployment.

and strives to order requests that depend on each other, imposing a partial order on requests. Sinfonia [27] and S-DUR [28] build a partial order by using a two-phase commit-like protocol to guarantee that requests spanning common partitions are processed in the same order at each partition. Spanner [29] orders requests within partitions using Paxos and across partitions using a protocol that computes a request's final timestamp from temporary timestamps proposed by the involved partitions.

Recovery: Recovery protocols often negatively affect a system's performance. Several optimizations can be applied to checkpointing and state transferring to minimize the overhead of recovery as we discuss next.

Some approaches have proposed to create checkpoints during the normal operation of a system, at the cost of halting normal command execution [8], [30], [31], [32]. If all replicas stop simultaneously, the system becomes unavailable to clients and reduces performance. In [12] processes schedule checkpoints at different intervals, and therefore, the system is always operational. As the operation of a quorum of processes is sufficient for their system to make progress, a minority of processes can be involved in a checkpointing while the other processes continue to operate. Another optimization is to use a *helper* process to take checkpoints asynchronously [33]. In this scheme, two threads, primary and the helper, execute concurrently. While the primary processes requests, the helper takes checkpoints periodically.

State transfer has its own performance issues. During

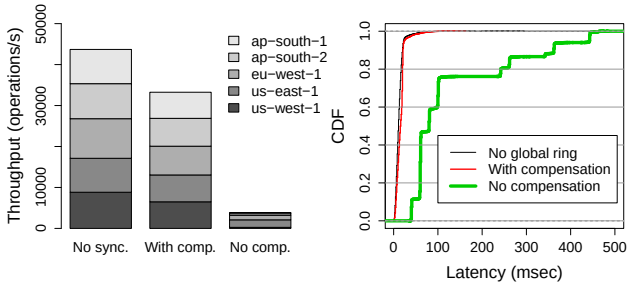

Figure 8. Impact of a global ring to local maximum throughput with and without latency compensated skip calculation.

state transfer, the source process is involved both in the execution of commands and the transmission of the state to the recovering process, which may hamper performance. To address this problem, state transfer can be delayed to a moment when the demand on the system is low enough that both the execution of new requests and the transfer of the state can be handled [13]. Another optimization is to reduce the amount of transferred information. Representing the state through efficient data structures [30], using incremental checkpoints [34], [33], or compressing the state are among these techniques. In [12], the authors propose a collaborative state transfer protocol to evenly distribute the transfer load across replicas. RAMCloud [35] is an in-memory storage system, where the data is also backed with persistent storage, such that the performance is not affected by the disk storage.

To recover the data fast RAMCloud relies on the collective force of thousands of servers. Differently from RAMCloud, our goal is to minimize application throughput disruption due to recovery, at the cost of a slower recovery procedure.

V. Lessons learned

In this section, we present some lessons we learned during the implementation and evaluation of Multi-Ring Paxos. We first share our experiences with high throughput, low latency Java applications, and then consider issues more specific to Multi-Ring Paxos.

A. High throughput and low latency in Java

Implementing Multi-Ring Paxos in Java comes with benefits and challenges. On the one hand, we can argue that Multi-Ring Paxos's code is easy to understand, maintain and extend. We have evidence of this observation from users of Multi-Ring Paxos in our research group. On the other hand, achieving high throughput in top-performing environments (e.g., 10 Gbps networks) and predictable low latency despite garbage collection is challenging.

Serialization. In the early versions of the URingPaxos library, serialization was a major performance problem. To avoid getting tied to a specific serialization library, we decided to keep our own internal objects and then translate to whatever object was required by the serialization library. This decision turned out to be problematic as, at high throughput, allocating all these extra objects caused a lot of garbage collection overhead. We finally settled on implementing our own serialization using Java's byte buffers and avoiding the creation of extraneous objects.

In-memory storage. Acceptors can be configured to use on-disk or in-memory storage. Using the in-memory storage (to avoid getting constrained by the performance of the disks), the acceptors have to keep enough data in memory to be able to handle the retransmission of recent messages (some seconds). In a 10 Gbit network, that adds up to GBytes of memory that are constantly being replaced. To avoid heavy garbage collection, we were keeping this data using a small library written in C, called through the Java Native Interface (JNI). It worked well but required a native, machine-dependent, library. Our current implementation achieves the same result by pre-allocating a large array of byte buffers.

Garbage collection and heap size. While garbage collection does not significantly impact average throughput, its effect is clearly visible on latency measurements. During our experiments, we observed that using smaller heap sizes resulted in smaller and more frequent GC pauses, leading to worse latency in average, but improving its standard deviation (i.e. short latency tail). On the other hand, larger heap sizes caused less frequent but longer GC pauses, resulting in better latency in average but larger standard deviation (i.e. long latency tail). This phenomenon can be observed in the garbage collection times CDF in Figures 2 and 3. The recovery experiment (Figure 4) also corroborates this idea: the large heap sizes we used incurred in GC pauses of up to a few seconds.

B. Protocol considerations

We now consider aspects specifically related to Multi-Ring Paxos: its ring topology and recovery.

Ring topology. While a ring topology helps achieving performance near nominal network capacity, it has the effect of propagating delays in a process to its successors. A pause in a single process (e.g. due to garbage collection or disk buffer flush) can cause the whole protocol to stop due to the serial propagation of messages in the ring. Whenever possible, processes should first forward messages to its successor before doing any local processing.

Recovery tradeoffs. Recovery involves many tradeoffs, which make the configuration of the protocol under high load difficult. While in the original procedure recovery boils down to installing the most recent checkpoint and fetching missing commands from acceptors, in the new procedure it involves caching new commands and waiting for a checkpoint that contains commands that precede the cached ones. The new method does not place acceptors under additional stress (to recover missing commands) but it increases memory usage and management at the replicas. Increased memory activity translates into new sources of overhead (e.g., garbage collection), which hurt performance.

The time needed for a checkpoint to be written to or read from stable storage introduces yet another tradeoff. In order to reduce the number of commands cached by a recovering replica, the time for an operational replica to store a checkpointed state and for the recovering replica to fetch the stored checkpoint should be short. Moreover, checkpoints must be created often. Creating a checkpoint, however, introduces overheads during normal execution (although this is minimized by copy-on-write optimizations).

VI. Conclusions

Internet-scale services rely on geographical distribution, data partitioning and replication to provide scalable performance and high availability. Building such systems poses many challenges, one of them being the need to carefully order requests among replicas and partitions. One way to cope with this problem is to use group communication primitives that encapsulate order requirements. Multi-Ring Paxos is a protocol in this category. Previous studies have considered Multi-Ring Paxos's performance in common environments. In this paper, we consider the protocol in more extreme conditions. While these conditions can be considered exceptional, many systems already need to face them. Therefore, understanding how Multi-Ring Paxos performs in such cases is important.

REFERENCES

[1] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty processor," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[2] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, pp. 51–59, June 2002.

[3] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Transactions on Computer Systems (TOCS)*, vol. 5, pp. 47–76, Feb. 1987.

[4] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems*, ch. 5, Addison-Wesley, 2nd ed., 1993.

[5] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys,*, vol. 36, pp. 372–421, Dec. 2004.

[6] P. J. Marandi, M. Primi, and F. Pedone, "Multi-ring paxos," in *DSN*, 2012.

[7] S. Benz, P. J. Marandi, F. Pedone, and B. Garbinato, "Building global and scalable systems with atomic multicast," in *Middleware*, 2014.

[8] L. Lamport, "The part-time parliament," *ACM (TOCS)*, 1998.

[9] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring paxos: A high-throughput atomic broadcast protocol," in *DSN*, 2010.

[10] L. Lamport, "Paxos made simple," *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, vol. 32, 2001.

[11] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[12] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia, "On the efficiency of durable state machine replication," in *ATC*, 2013.

[13] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *ATC*, 2010.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010.

[15] R. Jain, *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling*. New York: John Wiley and Sons, Inc., 1991.

[16] J. Fritzke, U., P. Ingels, A. Mostefaoui, and M. Raynal, "Fault-tolerant total order multicast to asynchronous groups," in *SRDS*, 1998.

[17] R. Guerraoui and A. Schiper, "Genuine atomic multicast in asynchronous distributed systems," *Theor. Comput. Sci.*, vol. 254, no. 1-2, pp. 297–316, 2001.

[18] L. Rodrigues, R. Guerraoui, and A. Schiper, "Scalable atomic multicast," in *ICCCN*, 1998.

[19] N. Schiper and F. Pedone, "On the inherent cost of atomic broadcast and multicast in wide area networks," in *ICDCN*, 2008.

[20] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton, "The Spread toolkit: Architecture and performance," tech. rep., Johns Hopkins University, 2004. CNDS-2004-1.

[21] D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia, "The totem multiple-ring ordering and topology maintenance protocol," *ACM*, May 1998.

[22] M. K. Aguilera and R. E. Strom, "Efficient atomic broadcast using deterministic merge," in *PODC*, 2000.

[23] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *SOSP*, 2007.

[24] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for wans," in *OSDI*, 2008.

[25] I. Moraru, D. G. Andersen, and M. Kaminsky, "Egalitarian paxos," in *SOSP*, 2012.

[26] N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine partial replication in wide area networks," in *SRDS*, 2010.

[27] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," in *SOSP*, 2007.

[28] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," in *DSN*, 2012.

[29] J. C. Corbett, J. Dean, and M. E. et al, "Spanner: Google's globally distributed database," in *OSDI*, 2012.

[30] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI*, 1999.

[31] J. Rao, E. J. Shekita, and S. Tata, "Using paxos to build a scalable, consistent, and highly available datastore," *Proceedings of the VLDB Endowment*, vol. 4, no. 4, pp. 243–254, 2011.

[32] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, P. Maniatis, *et al.*, "Zeno: Eventually consistent byzantine-fault tolerance.," in *NSDI*, 2009.

[33] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *SOSP*, 2009.

[34] M. Castro, R. Rodrigues, and B. Liskov, "Base: Using abstraction to improve fault tolerance," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 3, pp. 236–269, 2003.

[35] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, *et al.*, "The case for ramclouds: scalable high-performance storage entirely in dram," *ACM SIGOPS OSR*, vol. 43, no. 4, pp. 92–105, 2010.